Location Service

Jie Gao
Computer Science Department
Stony Brook University
Location service


Location service

- Geographical routing requires obtaining the location of the destination.

- What if the sensors move? How to update the location information?

- Internet: domain name server (DNS) translates user-friendly domain name (www.cnn.com) to machine-friendly IP address.
Centralized v.s. distributed location service

• Location server stores the mapping between locations and node IDs.
  – Centralized approach, single point of failure.
  – Communication bottleneck.
  – Location server might be far away.

• Distributed location servers: every node participates and acts as location servers for others.
Challenges

- **Problem 1**: each node need to know the location server of any node.
  - To update its own location info upon movement.
  - Query for the location of any other node.
- **Problem 2**: how to get to the location server?
  - We need a routing algorithm, say geographical routing.
- **Problem 3**: geographical routing requires the knowledge of destinations.
  - How to get the location of the location server?
  - Every node can be moving.
- **Problem 4**: location update upon node movement.
Proposal 1: Use GHT

- Each node holds a data = its location
- Use GHT to store the data by a hash function.

- Problem:
  - Distance insensitivity.
  - Frequent location update upon node movement.
Proposal 2: use double rulings

- Each node stores the data at the nodes on the producer curve.
- Problem: cost is too high for mobile nodes.
  - Producer curve has length $\sim \sqrt{n}$. 
Locate a mobile user

- **“Move”** operation:
  - inform system of new location

- **“Find”** operation:
  - Locate user at his current address.

- Distance-sensitivity: move to nearby locations or search for nearby users should be cheap.
  - Most moves are local
  - Most queries are local, too.
Model

- Connected, undirected, weighted graph $G$.
- Weight $w(e)$: cost on edge $e$.
- $\text{dist}(u, v)$: length of shortest path.
- Diameter $D(G)$: max distance of any two nodes in $G$.
- Address $\text{Addr}(x)$: current location of $x$.
- Assume an efficient routing scheme.
Model

• Consider a mixed sequence $\sigma$ of Move and Find operations.
  – $F(\sigma)$: subsequence of all Find operations in $\sigma$.
  – $M(\sigma)$: subsequence of all Move operations in $\sigma$.
• Cost: message transmissions.
• **Find-stretch**: cost $(F(\sigma))/\text{OPT}(F(\sigma))$
• **Move-stretch**: cost $(M(\sigma))/\text{OPT}(M(\sigma))$
• Goal: make Find-stretch and Move-stretch polylogarithmic in $n$. 
A distributed data structure

- Store pointers to locations of each user in various nodes.
- Pointers need to be updated as users move
- Allow some pointers to be inaccurate.

“Pointers at locations nearby to the user, whose update by the user is relatively cheap, are required to be more accurate, whereas pointers at distant locations are updated less often.”
Hierarchical directory server

- A hierarchy of $\delta = \lfloor \log D(G) \rfloor$ regional directories $RD_i$ ($1 \leq i \leq \delta$)

- $RD_i$ at level $i$ of the hierarchy enables a searcher to track any user within distance $2^i$.

- The address stored for user $x$ at $RD_i$ is called $i$-th level regional address $R_{addr_i}(x)$ --- where $x$ is currently expected to be.
Regional directory $RD_i$

- **Write$_i$(v)**
  - A node $v$ reports every user it hosts to all nodes in the write set.

- **Read$_i$(w)**
  - A searching node $w$ queries all nodes in some read set.

- **Read$_i$(w) and Write$_i$(v)** are guaranteed to intersect whenever $v$ and $w$ are within distance $2^i$ of each other.
$2^i$-regional matching

- $\text{Read}_i(w)$ and $\text{Write}_i(v)$ are guaranteed to intersect whenever $v$ and $w$ are within distance $2^i$ of each other.
2^i-regional matching

- Find operation invoked at node w will succeed at the lowest possible level enabled by the distance from w to v.
- At the highest level this operation will always succeed.

- Next question: When a node moves, update the addresses maintained at the location directory.
“Forwarding addresses”

• Whenever a user x moves a distance d, it update its regional directory on all levels.
  – Too expensive!
• Update only $\log d$ lowest levels
  – The lower the level, the more up-to-date the regional address
  – Low communication cost
  – But: the address $R_{addr_i}(x)$ might be old.
“Forwarding addresses”
The reachability invariant

- Define the tuple of regional addresses

\[ A(x) = \langle R_{Addr_1}(x), \cdots, R_{Addr_3}(x) \rangle \]

- \( R_{Addr_1}(x) = \text{Addr}(x) \) the true address.

- The reachability invariant: if at any time, \( R_{Addr_i}(x) \) stores a pointer \textbf{Forward}(x) to \( R_{Addr_j}(x) \) where \( j < i \).

- This may result in a long chain of forwarding pointers.
The proximity invariant

- The reachability invariant: if at any time, the distance travelled by x since the last time \( R_{\text{Addr}}(x) \) was updated satisfies
  \[ |\text{Migrate}_i(x)| \leq 2^{i-1} - 1 \]
- \( \text{Migrate}_i(x) \): the actual migration path from \( R_{\text{Addr}}(x) \) to its current location.
- A node currently hosting user x maintains Tuple of migration counters: \( C(x) = \langle C_1(x), \cdots, C_\delta(x) \rangle \)
  - \( C_i(x) \): distance travelled since the last update.
Updating regional addresses

Whenever user moves from a node s to a node t:
- Increase all migration counters $C_i$ by $\text{dist}(s,t)$. If the highest level counter $C_j$ reaches the upper limit ($2^{j-1} - 1$)
- Update the regional directory at levels 1 to $j$: set to $t$.
- Set forwarding pointer at $R_{\text{Addr}_{j+1}}(x)$ leading to $t$.
- Relocate user $x$ together with its tuples $A(x)$ and $C(x)$. 
Example

\[ A(\xi) = \langle x_4, x_4, x_3, x_2, x_2, x_1 \rangle \]

x4 stores:
\[ C(\xi) = \langle 0, 0, 3, 5, 5, 25 \rangle \]

\[ A(\xi) = \langle x_5, x_5, x_5, x_2, x_2, x_1 \rangle \]

x5 stores:
\[ C(\xi) = \langle 0, 0, 6, 6, 26 \rangle \]

\[ \text{dist}(x_4, x_5) = 1 \]
Discussion

• Proof of the cost in the full paper.

• Location service for one mobile user.

• What if all the nodes in the network are mobile?
GLS: Grid Location Service

- Last scheme: location update/query for a mobile user.
- All nodes are possibly mobile.
- Need to support queries for all nodes.
- Objective: balance the load, scalability.
GLS: Grid Location Service

- Each node is assigned a random ID in a circular space.
- Each node stores/updates its location information at a set of location servers, more at nearby regions, fewer at far away regions.
- Location query uses nothing beyond the ID.
- Two operations, FIND, UPDATE
Recursive partitioning

- Quad-tree partition: each node is inside a unique square on each level.

<table>
<thead>
<tr>
<th>Order 1 square</th>
<th>Order 2 square</th>
<th>Order 3 square</th>
<th>Order 4 square</th>
</tr>
</thead>
<tbody>
<tr>
<td>90 38</td>
<td>70 37</td>
<td>91 62 5 1</td>
<td>87 14 7 2</td>
</tr>
<tr>
<td>41 23 63</td>
<td>50 45 11 35</td>
<td>32 55 61 12</td>
<td>81 33 43 12</td>
</tr>
<tr>
<td>41 72</td>
<td>51 19 10 83</td>
<td>38 61 76 84</td>
<td>26 23 41 20</td>
</tr>
</tbody>
</table>

B: 17
Partitioning the world

Invariant: a node is located in exactly one square of each size (no overlapping)
An order-x square contains always 4 order-(x-1) squares
Location servers

- Node B’s location servers: Inside each sibling square on each level, choose B’s closest node.
- Node **closest** to B in ID space: node with least ID greater than B.
- Circular ID space: 2 is closer to 17 than 7 is.
Location queries

- A queries the location of B:
- A’s only information about B is the ID of B.
- A does not know who are B’s location servers.
- B even doesn’t know its location servers.
- How to implement location query?
Location queries

- A queries location of B:
- A stores location information for some other nodes.
- A send the request to the one that is closest to B, among those about which A has location information.
- Continue until hit one of B’s location servers.
- This works! Why?
Location queries

- Claim: the query visits the node closest to B in A’s order-i square.
- The query always goes to B’s closest node, as the covering scope increases.
- The correctness of the alg: when A’s order-i square contains B, the closest node is B itself.
- Proof by induction. It’s obvious for order-1 square.
Location queries

- Assume 21 is B’s closest node in A’s order-2 square ➔ no node is between 17 and 21 in order-1 square.
- Suppose a node X in A’s order-2 sibling square is between 17 and 21. By the replication rule, X picks 21 as its location server.
- 21 stores the location of all the nodes between 17 and 21 in sibling order-2 square, obviously the one closest to 17.
Inform/update location servers

- A can update its location server inside a square S without knowing its identify.
- A routes to a square with geographical routing.
- The first node in the square S performs a location query of A.
- The query ends up at a node closest to A, who is A’s location server!

Hidden assumption: the nodes in S have distributed their locations inside S!
The bootstrapping

- When the entire system is turned on, order-1 squares exchange their information with local protocol, then nodes recruit their order-2 location servers and so on.

- No flooding needed. The location service is constructed by geographical unicast routing only.
Take a rest and enjoy the beauty of this algorithm

- It solves location service problem by using geographical routing.
- More locality sensitive: a node acquires the location from a nearby server.
- Load balancing: location servers are spatially distributed.
- Simple rule, simple construction and maintenance.
- Worst-case query behavior is not bounded, however. 😞
Handle mobility

• For the geographic forwarding protocol:
  – Refresh neighbor information using Hello messages, timeouts.

• Updating location servers when a node moves:
  – Not every time—that would be too much communication.
  – Rather, they use the Awerbuch-Peleg idea: Update order-i servers when you’ve moved $2^{i-1}$ distance.
  – Thus, updates are sent more frequently to local servers than more distant ones.
Handle mobility

• Invalidating location table entries:
  – They have a timeout value.
  – Even when a node doesn’t move, it thus has to periodically send its location to its servers, in order to refresh the location info.

• Caching:
  – Nodes can cache locations they hear about, to use in sending data via geographic routing—but these entries aren’t put into the same table used by FIND—recall that it’s important that certain entries NOT be in these tables.
Handle mobility

• Two types of failures
  – A node receives a query but does not know the location of any node with an ID closer. Location update is not soon enough.
  
  – A location server forwards a packet to the next node’s square but the node is not there. When a node moves to a nearby square, it leaves a forwarding pointer.

  – Can be improved.
Follow-up work

Main idea of LLS

- Handle node mobility
- Worst-case guarantee of FIND & MOVE
- Recall for Grid:
  - Cost for MOVE can be high, if a node crosses quad-tree boundaries.
  - Cost for FIND can be high, if a node is just at the neighboring square separated far in the hierarchy.
Each node defines a square partitioning centered at itself.

Spiral: store the location at the corners of the squares.

MOVE is heavy --- erase the old spiral and draw the new spiral.

FIND is efficient: use its spiral.

Cost(FIND) = O(d).
LLS --- improve MOVE behavior

• Store location in the 8 squares surrounding each square.
  – Some flexibility to allow “lazy” update.
  – When a node moves inside the 9 squares, nothing happens.
  – When a node moves outside the 9 squares, it has moved “sufficiently” far away from its previous location and can drag the 9 squares with it.
  – Amortized update cost is low.
LLS performance

- MOVE: $O(d \log d)$
- FIND: $O(d)$
- Amortized cost.
What is next?

- The hierarchies use location-based hierarchies, quadtree, etc.
- In a graph setting, we use a hierarchy of landmarks.
- Landmark-based routing and location service.
Landmark hierarchy

• Nodes are organized as landmarks of different levels.
• All nodes are level-0 landmarks.
• Some nodes are selected as level-1 landmarks and so on.
  – Covering: each level i-1 landmark is within distance $2^i$ from at least one level i landmark and select one as its parent;
  – Packing: every two level i landmarks are of distance $2^i$ away.
1st level landmarks
2nd level landmarks
3\textsuperscript{rd} level landmarks
Landmark hierarchy

- # levels: h = log n.
- A node is within distance $2^{i+1}$ from its level i ancestor.
  - The distances along the path to the ancestor is at most $1 + 2 + 4 + \ldots + 2^i = 2^{i+1} - 1$
- A landmark at level i has a cluster = its descendants.
Landmark hierarchy

- Each landmark has $O(1)$ children if the graph has constant doubling dimension.
  - Recall constant doubling dimension: each ball of radius $r$ can be covered by $\beta$ balls of radius $r/2$.
  - Packing argument: a landmark $u$ at level $i$ has $k$ children within distance $r$. The ball at this landmark can be covered by $\beta^3$ balls of radius $r/8$. Call them $B$. Take balls at $u$’s children with radius $r/8$, these balls intersect disjoint sets of balls in $B$. Thus there are $O(1)$ children.
Example
Addresses/labels/virtual coordinates

- Each node keeps its position in the tree.
- A landmark at level $i$ has a tuple of length $h$ including all its ancestors from level $i$ up and 0 from level $i$ down.
How to route?

- We can route on the landmark hierarchy (which is a tree).
- A node \( p \) routes to the lowest common ancestor of \( p, q \).
- One can identify the lowest common ancestor from their addresses.
- But how to route down the tree?

\[
\begin{align*}
p &= (\textcolor{blue}{\bigcirc} \textcolor{green}{\bigcirc} \textcolor{red}{\bigcirc} ) \\
q &= (\textcolor{blue}{\bigcirc} \textcolor{green}{\bigcirc} \textcolor{red}{\bigcirc} )
\end{align*}
\]
Routing

• To aid routing, we add cross-branch links.
  – A node maintains routing table to reach the cluster of a landmark at level i if it is within distance $\alpha 2^{i+1}$ from that cluster.
  – Landmark at level i floods with TTL=$((\alpha + 1)2^{i+1})$. 
How to route?

• A node \( p \) routes to the **lowest level cluster** that contains \( q \).
  – Can possibly be lower than their common ancestor.
  – Can be identified with \( q \)'s address.

• At each step, route to a **lower level cluster** containing \( q \).

\[ p = (\textcolor{blue}{\textcircled{1}}, \textcolor{green}{\textcircled{2}}, \textcolor{red}{\textcircled{3}}, \textcolor{black}{\textcircled{4}}) \]
\[ q = (\textcolor{blue}{\textcircled{1}}, \textcolor{green}{\textcircled{2}}, \textcolor{red}{\textcircled{3}}, \textcolor{black}{\textcircled{4}}) \]
How to route?

- A node p routes to the lowest level cluster w (at level i) that contains q.
- First leg to w's cluster: at most $\alpha 2^{i+1}$.
- Note that w is also at level i-1.
- Since w at level i-1 is not a neighboring cluster, $|pq|>\alpha 2^i$. Thus the first leg is at most $2|pq|$. The next leg is halved in length.
- Total length is at most $4|pq|$.

$p=$

$q=$
Data-centric routing, aka, location service

- Each node \( p \) stores its location at a hashed node inside the cluster of all the neighboring clusters.
- FIND: search the clusters of \( q \)'s ancestors.
Data-centric routing, aka, location service

- Example in a quad-tree setting.
FIND cost

- q finds p in its ancestor cluster w at level i.
- w at level i-1 is not a neighboring cluster of p. That means, $|pq| > \alpha 2^i$.
- q's cost is at most $\alpha 2^{i+2}$, thus at most $4|pq|$.
Open issues on location service

• Make use of node mobility?
  – When two nodes pass by, they keep each other’s info.

• Security issue with location service?